CS5412: THE BASE METHODOLOGY VERSUS THE ACID MODEL IN AN OVERLAY

Lecture IX

Ken Birman

Recall the overlay network idea

- For example, Dynamo, Chord, etc
- Basically, a key-value storage system spread over a big pool of nodes inside Facebook or some other cloud platform. Lookups are fast (1-hop if you have a list of the nodes in the DHT, as they do at Facebook or Amazon or Google).
- Very popular and important, used extensively in the cloud. ***Perhaps the single most important enabler for scalability in today's cloud systems!***

Question we'll focus on today

- Suppose that one application does a "put" to store something in the DHT
  - For example, it stores a new profile picture for Ken
- Will some other application be sure to see the new version if it issues a subsequent "get"?
  - Sounds reasonable, but some form of locking would be required to implement this behavior
  - We'll see that this is an issue in today's cloud and in fact the prevailing approach just says no to locking!

More general version of this question

- Those of you taking Al Demers' course will be thinking about massive databases

- They also have a model focused on updates and reads and the same issue arises in transactional applications

- So: should the cloud try to be transactional in these giant, super-fast key-value storage structures?

Methodology versus model?

- Today's lecture is about an apples and oranges debate that has gripped the cloud community
  - A methodology is a "way of doing" something
    - For example, there is a methodology for starting fires without matches using flint and other materials
  - A model is really a mathematical construction
    - We give a set of definitions (i.e. fault-tolerance)
    - Provide protocols that provably satisfy the definitions
    - Properties of model, hopefully, translate to application-level guarantees

The ACID model

- A model for correct behavior of databases
- Based on the concept of transaction
- A **transaction** is a sequence of operations on database or data store that form a single unit of work.
  - Operations: reads or writes
- A transaction transforms a database from one consistent state to another
  - During execution the database may be inconsistent
- All operations must succeed; otherwise transaction fails

ACID as a methodology

Body of the transaction performs reads and writes, sometimes called queries and updates

- We teach it all the time in our database courses
- Students write tra

Begin signals the start of the transaction

```
Begin
    let employee t = Emp.Record("Tony");
    t.status = "retired";
    ∀ customer c: c.A
            c.AccountF
Commit;
```

Commit asks the database to make the effects permanent. If a crash happens before this, or if the code executes **Abort**, the transaction rolls back and leaves no trace

- System executes this code in an all-or-nothing way

ACID model properties

- Issues:
  - Concurrent execution of multiple transactions
  - Recovery from failure
- Name was coined (no surprise) in California in 60's
  - **Atomicity**: Either all operations of the transaction are properly reflected in the database (commit) or none of them are (abort).
  - **Consistency**: If the database is in a *consistent* state before the start of a transaction it will be in a *consistent* state after its completion.
  - **Isolation**: Effects of ongoing transactions are not visible to transaction executed concurrently. Basically says "we'll hide any concurrency"
  - **Durability**: Once a transaction commits, updates can't be lost or rolled back

ACID example

- Transaction to transfer $10000 from account A to account B:
  1. read(A)
  2. A := A – 10000
  3. write(A)
  4. read(B)
  5. B := B + 10000
  6. write(B)
- Consistency requirement – the sum of A and B is unchanged by the execution of the transaction.
- Atomicity requirement — if the transaction fails after step 3 and before step 6, the system should ensure that its updates are not reflected in the database, else an inconsistency will result.

ACID example continued…

- Durability requirement — once the user has been notified that the transaction has completed (i.e., the transfer of the $10000 has taken place), the updates to the database by the transaction must persist despite failures.
- Isolation requirement — if between steps 3 and 6, another transaction is allowed to access the partially updated database, it will see an inconsistent database (the sum A + B will be less than it should be).
  Can be ensured trivially by running transactions serially, that is one after the other. However, executing multiple transactions concurrently has significant benefits, as we will see.

Why ACID is helpful

- Developer doesn’t need to worry about a transaction leaving some sort of partial state
  - For example, showing Tony as retired and yet leaving some customer accounts with him as the account rep
- Similarly, a transaction can’t glimpse a partially completed state of some concurrent transaction
  - Eliminates worry about transient database inconsistency that might cause a transaction to crash
  - Analogous situation: thread A is updating a linked list and thread B tries to scan the list while A is running

Implementation considerations

- Atomicity and Durability:
  - Shadow-paging (copy-on-write):
    - updates are applied to a partial copy of the database,
    - the new copy is activated when the transaction commits.
  - Write-ahead logging (in-place):
    - all modifications are written to a log before they are applied.
    - After crash: go to the latest checkpoint, replay log.

Implementation considerations

- Isolation:
  - Concurrency control mechanisms: determine the interaction between concurrent transactions.
  - Various levels:
    - Serializability
    - Repeatable reads
    - Read committed
    - Read uncommitted

ACID another example

- Imagine the following set of transactions:
  - T0: Employee.Create("Sally", "Intern", Intern.BaseSalary);
  - T1: Sally.salary = Sally.salary*1.05%
  - T2: Sally.Title =" Supervisor";
    Sally.Salary = Supervisor.BaseSalary;
  - T3: Print(SUM(e.Salary where e.Title="Intern")/ Count(e WHERE e.Title == "Intern"));
    Print(SUM(e.Salary where e.Title="Supervisor")/ Count(e WHERE e.Title == "Supervisor"))

ACID another example

- What happens if order changes:
  - T0, T1, T2, T3 vs. T0, T2, T1, T3 vs. T0, T3, T1, T2
- Which outcome is 'correct'?
- Is there a case where multiple outcomes are valid?
- What ordering rule needs to be respected for the system to be an ACID database?

Serial and Serializable executions

- A “serial” execution is one in which there is at most one transaction running at a time, and it always completes via commit or abort before another starts
- “Serializability” is the “illusion” of a serial execution
  - Transactions execute concurrently and their operations interleave at the level of the database files
  - Yet database is designed to guarantee an outcome identical to some serial execution: it masks concurrency
  - In past they used locking; these days “snapshot isolation”
  - Will revisit this topic in April and see how they do it

Implementation considerations

- □ Consistency: A state is consistent if there is no violation of any *integrity constraints*

- □ Consistency is expressed as predicates data which serves as a precondition, post-condition, and transformation condition on any transaction

- □ Application specific

- □ Developer's responsibility

All ACID implementations have costs

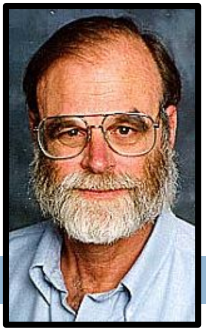
- Locking mechanisms involve competing for locks and there are overheads associated with how long they are held and how they are released at Commit

- Snapshot isolation mechanisms using locking for updates but also have an additional *version* based way of handing reads
  - Forces database to keep a history of each data item
  - As a transaction executes, picks the versions of each item on which it will run

- So… there are costs, not so small

Dangers of Replication

[The Dangers of Replication and a Solution . Jim Gray, Pat Helland, Dennis Shasha. Proc. 1996 ACM SIGMOD.]



- Investigated the costs of transactional ACID model on replicated data in "typical" settings
  - Found two cases
    - Embarrassingly easy ones: transactions that don't conflict at all (like Facebook updates by a single owner to a page that others might read but never change)
    - Conflict-prone ones: transactions that sometimes interfere and in which replicas could be left in conflicting states if care isn't taken to order the updates
  - Scalability for the latter case will be *terrible*
- Solutions they recommend involve sharding and coding transactions to favor the first case

Approach?

- They do a paper-and-pencil analysis
  - Estimate how much work will be done as transactions execute, roll-back
  - Count costs associated with doing/undoing operations and also delays due to lock conflicts that force waits
- Show that even under very optimistic assumptions slowdown will be $O(n^2)$ in size of replica set (shard)
- If approach is naïve, $O(n^5)$ slowdown is possible!

This motivates BASE

[D. Pritchett. BASE: An Acid Alternative. ACM Queue, July 28, 2008.]

- Proposed by eBay researchers
  - Found that many eBay employees came from transactional database backgrounds and were used to the transactional style of “thinking”
  - But the resulting applications didn’t scale well and performed poorly on their cloud infrastructure
- Goal was to guide that kind of programmer to a cloud solution that performs much better
  - BASE reflects experience with real cloud applications
  - “Opposite” of ACID

A “methodology”

- BASE involves step-by-step transformation of a transactional application into one that will be far more concurrent and less rigid
  - But it doesn’t guarantee ACID properties
  - Argument parallels (and actually cites) CAP: they believe that ACID is too costly and often, not needed
  - BASE stands for “**Basically Available Soft-State Services with Eventual Consistency**”.

Terminology

- **Basically Available:** Like CAP, goal is to promote rapid responses.
  - BASE papers point out that in data centers partitioning faults are very rare and are mapped to crash failures by forcing the isolated machines to reboot
  - But we may need rapid responses even when some replicas can't be contacted on the critical path

Terminology

- **Basically Available:** Fast response even if some replicas are slow or crashed
- **Soft State Service:** Runs in first tier
  - Can’t store any permanent data
  - Restarts in a “clean” state after a crash
  - To remember data either replicate it in memory in enough copies to never lose all in any crash or pass it to some other service that keeps “hard state”

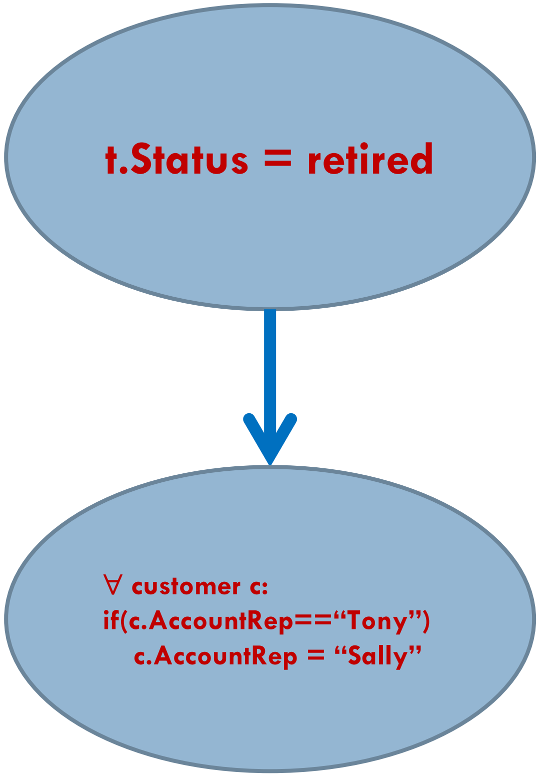
Terminology

- **Basically Available:** Fast response even if some replicas are slow or crashed
- **Soft State Service:** No durable memory
- **Eventual Consistency:** OK to send "optimistic" answers to the external client
  - Could use cached data (without checking for staleness)
  - Could guess at what the outcome of an update will be
  - Might skip locks, hoping that no conflicts will happen
  - Later, if needed, correct any inconsistencies in an offline cleanup activity

How BASE is used

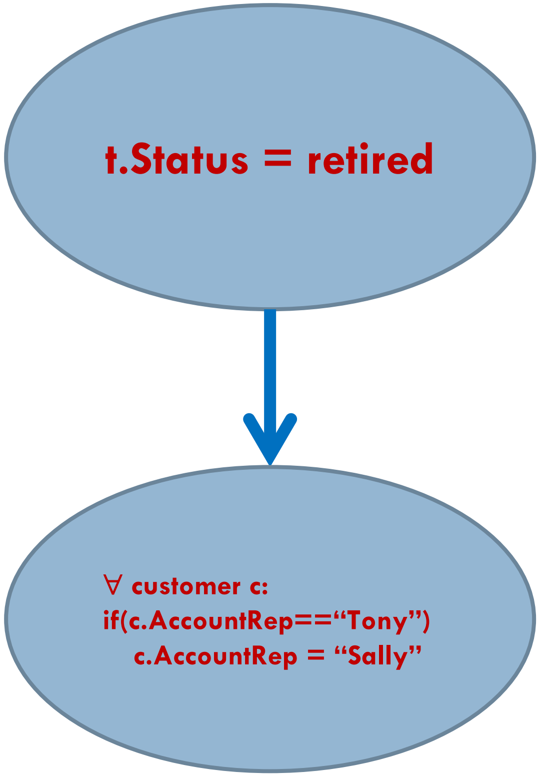
- Start with a transaction, but remove Begin/Commit
  - Now fragment it into "steps" that can be done in parallel, as much as possible
  - Ideally each step can be associated with a single event that triggers that step: usually, delivery of a multicast
- Leader that runs the transaction stores these events in a "message queuing middleware" system
  - Like an email service for programs
  - Events are delivered by the message queuing system
  - This gives a kind of all-or-nothing behavior

Base in action



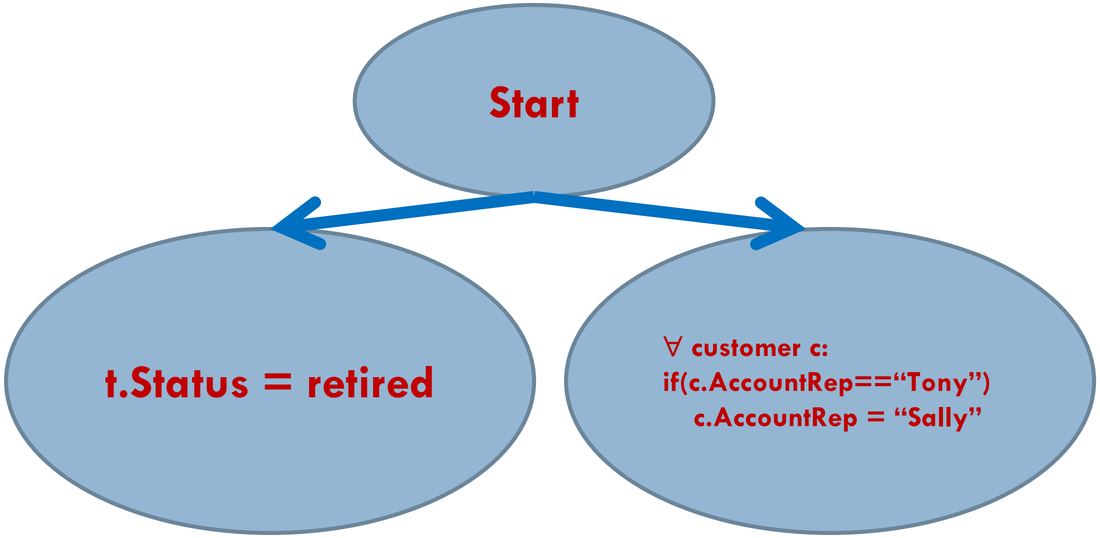
```
Begin
    let employee t = Emp.Record(“Tony”);
    t.status = “retired”;
    ∀ customer c: c.AccountRep==“Tony”
            c.AccountRep = “Sally”
Commit;
```

Base in action



t.Status = retired

```
∀ customer c:
if(c.AccountRep==“Tony”)
    c.AccountRep = “Sally”
```



Start

t.Status = retired

```
∀ customer c:
if(c.AccountRep==“Tony”)
    c.AccountRep = “Sally”
```

More BASE suggestions

- Consider sending the reply to the user before finishing the operation
- Modify the end-user application to mask any asynchronous side-effects that might be noticeable
  - In effect, “weaken” the semantics of the operation and code the application to work properly anyhow
- Developer ends up thinking hard and working hard!

Before BASE… and after

- Code was often much too slow, and scaled poorly, and end-user waited a long time for responses

- With BASE
  - Code itself is way more concurrent, hence faster
  - Elimination of locking, early responses, all make end-user experience snappy and positive
  - But we do sometimes notice oddities when we look hard

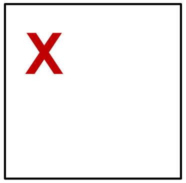
BASE side-effects

- Suppose an eBay auction is running fast and furious
  - Does every single bidder necessarily see every bid?
  - And do they see them in the identical order?

- Clearly, everyone needs to see the winning bid

- But slightly different bidding histories shouldn't hurt much, and if this makes eBay 10x faster, the speed may be worth the slight change in behavior!

BASE side-effects

- Upload a YouTube video, then search for it
  - You may not see it immediately
- Change the “initial frame” (they let you pick)
  - Update might not be visible for an hour
- Access a FaceBook page when your friend says she’s posted a photo from the party
  - You may see an 

BASE in action: Dynamo

- Amazon was interested in improving the scalability of their shopping cart service

- A core component widely used within their system
  - Functions as a kind of key-value storage solution
  - Previous version was a transactional database and, just as the BASE folks predicted, wasn't scalable enough
  - Dynamo project created a new version from scratch

Dynamo approach

- They made an initial decision to base Dynamo on a Chord-like DHT structure

- Plan was to run this DHT in tier 2 of the Amazon cloud system, with one instance of Dynamo in each Amazon data center and no "linkage" between them

- This works because each data center has "ownership" for some set of customers and handles all of that person's purchases locally.

The challenge

- Amazon quickly had their version of Chord up and running, but then encountered a problem

- Chord isn't very "delay tolerant"
  - So if a component gets slow or overloaded, Chord was very impacted
  - Yet delays are common in the cloud (not just due to failures, although failure is one reason for problems)

- Team asked: how can Dynamo tolerate delay?

Idea they had

- Key issue is to find the node on which to store a key-value tuple, or one that has the value

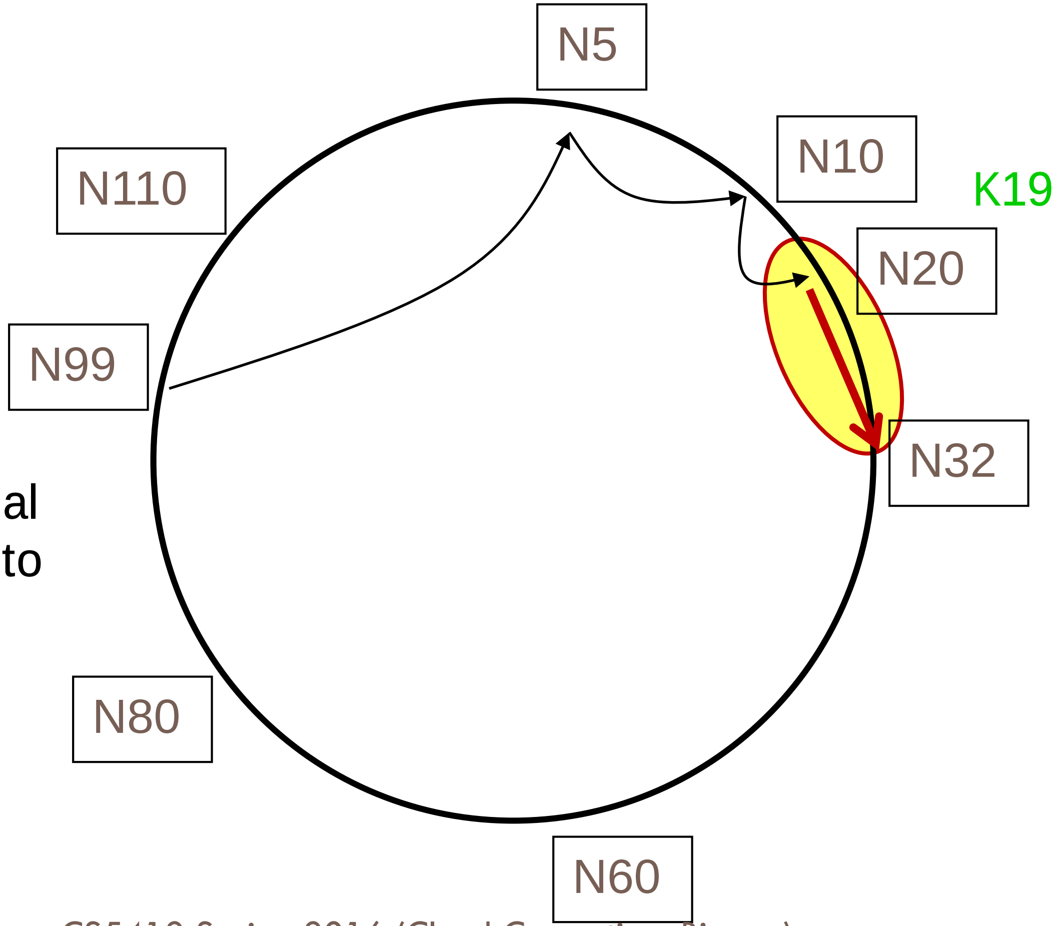
- Routing can tolerate delay fairly easily
  - Suppose node K wants to use the finger to node $K+2^i$ and gets no acknowledgement
  - Then Dynamo just tries again with node $K+2^{i-1}$
  - This works at the "cost" of slight stretch in the routing path in the rare cases when it occurs

What if the actual “home” node fails?

- Suppose that we reach the point at which the next hop should take us to the owner for the hashed key
- But the target doesn’t respond
  - It may have $k+2^{i-1}$ crashed, or have a scheduling problem (overloaded), or be suffering some kind of burst of network loss
  - All common issues in Amazon’s data centers
- Then they do the Get/Put on the *next node that actually responds* even if this is the “wrong” one!

Dynamo example: picture

Lookup(K19):
API is designed to look transactional but actually maps to Dynamo DHT



Dynamo example in pictures

- Notice: Ideally, this strategy works perfectly
  - Recall that Chord normally replicates a key-value pair on a few nodes, so we would expect to see several nodes that “know” the current mapping: a shard
  - After the intended target recovers the repair code will bring it back up to date by copying key-value tuples
- But sometimes Dynamo jumps beyond the target “range” and ends up in the wrong shard

Consequences?

- If this happens, Dynamo will eventually repair itself
  - … But meanwhile, some slightly confusing things happen

- Put might succeed, yet a Get might fail on the key

- Could cause user to "buy" the same item twice
  - This is a risk they are willing to take because the event is rare and the problem can usually be corrected before products are shipped in duplicate

Dynamo-DB

- When Dynamo was introduced, it had a quick uptake at Amazon but then stalled
  - For most key-value uses it was perfect and easily adopted
  - But for applications coded against Oracle's SQL interfaces that expected transactions, too much recoding was needed
- Dynamo-DB builds a kind of fake transactional SQL API over Dynamo
  - It doesn't guarantee ACID, but was close enough to what the database people needed and wanted to be adopted.

Werner Vogels on BASE

- He argues that delays as small as 100ms have a measurable impact on Amazon's income!
  - People wander off before making purchases
  - So snappy response is king

- True, Dynamo and Dynamo-DB have weak consistency and may incur some delay to achieve consistency
  - There isn't any real delay "bound"
  - But they can hide most of the resulting errors by making sure that applications which use Dynamo don't make unreasonable assumptions about how Dynamo will behave

Conclusion?

- BASE is a widely popular alternative to transactions
  - Used (mostly) for first tier cloud applications
  - Weakens consistency for faster response, later cleans up
  - eBay, Amazon Dynamo shopping cart both use BASE

- Later we'll see that strongly consistent options do exist
  - In-memory chain-replication
  - Send+Flush using Isis^2
  - Snapshot-isolation instead of full ACID transactions

- Will look more closely at latter two in a few weeks